(2)

CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: Final Report - Comprehensive Race Controls: A Versatile Scheduling Mechanism for Real-Time Applications

CIN: C08 092KU 0001 00

23 February 1990

90-2298

COMPREHENSIVE RACE CONTROLS: A VERSATILE SCHEDULING MECHANISM FOR REAL-TIME APPLICATIONS

FINAL REPORT

PREPARED FOR:

U.S. Army HQ CECOM
Center for Software Engineering
Advanced Software Technology
Fort Monmouth, NJ 07703-5000

PREPARED BY:

Tzilla Elrad
Illinois Institute of Technology
IIT Center
Chicago, IL 60616

DATE

February 9, 1990

Delivery Order 1263
Scientific Services Program

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Ada is designed for real-time and hard real-time concurrent applications. Real-time systems usually exhibit traits such as intelligence, adaptability, and a highly dynamic behavior. They frequently contain periodic, time-critical processes which have hard deadlines for completion. Other real-time systems consist of a set of prioritized processes where the priority of the process dictates the execution sequence. Therefore, a real-time computing system depends not only on the correctness of results but also on the timeliness of the results. For example, a missed deadline in a hard real-time system means failure. So the objective of real-time programming is to meet the timing requirement imposed upon each task. And yet other highly adaptable real-time systems must react to their external environment to determine the execution sequence by monitoring data, through acquisition or analysis, and communications. Therefore, another objective of real-time programming is to adapt external environment requirements upon the execution of each task. These objectives are achieved through the use of a scheduler which controls and resolves task execution and intertask communication conflicts. Schedulers must be preemptive and priority driven to react to the real-time system's highly dynamic and adaptive nature. These system characteristics also encourage the use of dynamic scheduling policies as opposed to the use of static scheduling policies commonly found in today's real-time systems. The constraints imposed by real-time applications require that scheduling be accomplished in such a way that the system is understandable, maintainable, predictable and adaptable.

The Explicit Comprehensive Set of Race Controls represent scheduling controls that are essential for the design and implementation of predictable, understandable, maintainable and adaptable real-time systems. These controls are language constructs that allow for the natural expression of scheduling controls consistently across the range of all possible controls on indeterminate behavior.

Ada represents a rich and powerful tasking model which offers an alternative design to the cyclical executive model. Ada's tasking model consists of tasks which are competing with other tasks for execution and intertask communication. However, nondeterministic constructs such as the **selective wait** apply fairness semantics over predictability semantics and; therefore, are not sufficiently predictable. Hence, the major concern of real-time programming using the Ada tasking model is the controlling of the high degree of nondeterminism. To achieve more predictable real-time programs, the scheduling of alternative events assumes a major role. The scheduler performs the important function of resolving these conflicts. The Ada tasking model is a natural model for applications which must adhere to an indeterminate environment. The opposing position is that the Ada tasking model is inadequate for real-time applicati [illegible]ause hard deadline constraints can not be guaranteed for systems exhibiting indeterminate behavior. This situation occurs for two reasons. One is the inconsistency in the application of scheduling controls across the range of possible controls within the Ada environment. The other is due to an incompleteness of the Ada language to express the control of indeterminate behavior. The Explicit Comprehensive Set of Race Controls are consistently defined and have the capability to express controls on indeterminate behavior. Ada could be further enriched by adding these controls to the language. This could be done because the Explicit Comprehensive Set of Race Controls conforms to the Ada design philosophy.

Tasking models should be conducive to the design and implementation of concurrent real-time applications. However, there exists a lack in design methodologies for real-time systems. Research [8, 13] in this area is currently in progress but has not been extended to the tasking model. Most real-time system design today is ad hoc using design methods that result from the cyclical executive model. There is a need for real-time system design methodologies that incorporate the timing and reactive constraints of a system as a system specification from the beginning of the design process. To satisfy real-time system specifications, scheduling policies should be designed into an application so that the system is not less predictable than the environment in which they are embedded. In addition, the indeterminate behavior characteristic to the problem domain must be simply and naturally expressed in the solution domain of the tasking model. Ada is characterized as a "design language" suitable for expressing a solution throughout the entire life cycle of a software project [2]. The Explicit Comprehensive Set of Race Controls conforms to these requirements. They can be incorporated into the specification phase of a project and then can be easily translated into the program development phase.

The Explicit Comprehensive Set of Race Controls exhibit qualities that are recommended in the Ada 9X revision process [1, 12]. Language Issue 62 (LI62) formulated by the Ada Language Issues Working Group (ALIWG) states that Ada 9X should allow for "increased control over task scheduling." The specific requirement is that the user should be able to control the method by which tasks are scheduled. The Parallel/Distributed Systems Working Group of the Ada 9X Project Requirement Workshop developed the requirement for adaptive scheduling that the Ada language shall:

1. Not prohibit scheduling by context, which may be dynamic,
2. Provide mechanisms for scheduling by multiple characteristics, including user defined characteristics, and
3. Support different paradigms in different parts of the system.

The Explicit Comprehensive Set of Race Controls gives dynamic scheduling control to the user and is flexible enough to represent many different scheduling paradigms distributed throughout the system.

This report presents the framework for nondeterminism, scheduling and its control with respect to a general tasking model in sections II, III and IV. Section V applies these notions to the Ada tasking model. Section VI discusses a scheduling anomaly that occurs in Ada due to conflicts caused by the inconsistency in the application of race controls. Section VII outlines a current research proposal from the SEI for race control. Section VIII presents the Comprehensive Set of Race Controls including applications and a comparison with the SEI proposal.

II. NONDETERMINISM WITHIN A GENERAL TASKING MODEL

Nondeterminism exists within a general tasking model at the program level, the task level and the entry level. At the program level, a new task must be scheduled whenever a task is suspended or terminated. The scheduler selects one task out of eligible tasks. At the task level, the semantics of the language usually requires that one of the open alternatives of the selective wait construct be chosen. At the entry level, a number of calls may be pending for that particular entry. A choice has to be made.

A method for implementing some control of indeterminate behavior is through the use of FIFO scheduling: tasks will be scheduled in the order in which they become ready, the alternative selected will be the first open alternative encountered and entry calls will be accepted in the order that they occur. FIFO queues are simple, fair and efficient to implement and are adequate for many concurrent applications. However, they are inadequate for real-time and hard real-time applications that require tight deadlines and changing environmental conditions. FIFO scheduling does not respond to the urgency of certain events whether they be tasks, services or communications. In order to design real-time systems that reflect real-time constraints, scheduling controls are required at each level to create more deterministic programs which are suitable for a real-time environment.

Nondeterminism has been identified within and associated with the level at which it occurs within the general tasking model. This association enabled us to classify the scheduling controls as to the level at which they function.

III. THE CLASSIFICATION OF SCHEDULING CONTROLS

The set of all possible scheduling controls used by a language is termed the **Comprehensive Scheduling Controls**. Figure 1 illustrates the Comprehensive Scheduling Controls hierarchy. The **Comprehensive Scheduling Controls** are split into two classes: **Availability Controls** and **Race Controls**.

Availability Controls are those controls which enable or disable a nondeterministic choice within the alternative construct of a concurrent language. They determine which alternatives will be available for selection. Or, alternatively, availability controls are used to bar events for which the decision making unit is not ready yet. The availability of any of the alternatives for selection is derived from certain constraints applied to each alternative, such as Boolean expressions, communication readiness or any combination of such constraints. Those alternatives in which all the constraints are satisfied are said to be **open** and available for selection, otherwise the alternatives are said to be **closed** and unavailable

```
            Comprehensive Scheduling Controls
                     /                \
                    /                  \
                   /                    \
      Availability Controls          Race Controls
              |                           |
              |                           |
              |                           |
        Consensus Control          Priority Control
        Private Control            Preference Control
        Mutual Control             Forerunner Control
        Hybrid Control
```

Figure 1. Scheduling Control Classifications

for selection. Availability Controls are subclassified as **Consensus Control, Private Control, Mutual Control** and **Hybrid Control** to reflect the different constraints that may be applied to an alternative and are defined as follows.

Consensus Control: The capacity to enable or disable an alternative based on pending communication requests.
Private Control: The capacity to enable or disable an alternative based on a Boolean expression determined by the local state of the task.
Mutual Control: The capacity to enable or disable an alternative based on a Boolean expression determined by both the local state of the task and the caller state.
Hybrid Control: is the capacity to enable or disable an alternative based upon any combination of consensus, private, or mutual controls.

Availability controls are not the focus of this report and are included here for completeness only.

Race Controls are those which control the scheduling of task, alternative or entry call whenever a choice situation occurs for that type in a concurrent program. Tasks are racing to be scheduled when a resource becomes free, open alternatives are racing to be selected when an alternative construct is executed and entry calls are racing to be accepted when an intertask communication occurs. To reflect each of these races, race controls are subclassified as **Priority Control, Preference Control** and **Forerunner Control** and are defined as follows:

Priority Control: The prioritization of tasks for program level scheduling.
Preference Control: The prioritization of alternatives for task level scheduling.
Forerunner Control: The prioritization of entry calls for entry level scheduling.

In describing the difference between Availability Controls and Race Controls, it is helpful to note that Availability Controls determine what **could** be done next whereas Race Controls determine what **should** be done next.

Race controls are primarily used to select an event from available choices and to prioritize the events. The necessity for race controls is crucial within real-time systems as real-time systems rely heavily on the prioritization of events.

IV. THE TYPES OF RACE CONTROLS

1. Lack of Control

Lack of control means no control over the races. Control is implemented as part of the compiler or runtime system. There exists no language specification to control a race. A rule in which an alternative is selected arbitrarily in a **selective wait** is an example of lack of control.

Lack of control stands alone as a characteristic because it in itself represents one complete type of scheduling possibility. The characteristics to be discussed in the following subsections imply some type of control and can be combined to form many scheduling possibilities.

2. Implicit vs. Explicit Control

Race controls that are **implicit** are language specifications that are handled exclusively by the scheduler. The language specifies rules on how each race will be resolved and the scheduler must conform to these rules to resolve each race. For example, the language may specify that the entry queues must be ordered on a FIFO basis. The scheduler must then be implemented to satisfy this rule.

Race controls that are **explicit** are programmer specifications. An explicit race control is handled exclusively by the programmer through a mechanism that generates the desired control. This simplifies the scheduler since any race control handled explicitly does not need to be incorporated into the scheduler. An example of explicit control is the implementation of Priority Control in Ada. The **pragma priority** statement acts as a directive to the compiler to set the priority of a task at a fixed value.

There is a hierarchy of control from a lack of control to implicit control to explicit control. Figure 2 illustrates this with respect to the entry queue. As one moves through the hierarchy, one gains more control over the environment producing more predictable systems by transferring the responsibility of control closer and closer to the user.

3. Static vs. Dynamic Control

Race controls can be either static or dynamic. A static race control fixes the priority of its entity (task, alternative, or entry) at compile time and no change in the entity's priority may *occur at runtime*. A dynamic race control allows the priority of an entity to be modified at runtime.

Statically controlled races are easy to implement but they

```
                    Explicit Control
     ↑               Programmer Specification
     |                      Example:  Entry queue controlled by "by"
     | C                                construct in Concurrent C
M    |  o
 o   |   n          Implicit Control
  r  |    t            Language Specification
   e |     r                Example:  Entry queue handled in FIFO
     |      o                           order in Ada
     |       l
     |              Lack of Control
     |                 Compiler or Run-time system implementation
                            Example:  Entry queue handled by an
                                        arbitrary selection
```

Figure 2. Hierarchy of Control

are not satisfactory for applications that must reflect volatile or adaptable environments where priorities need to be constantly changed. Therefore, static controls result in non-adaptive scheduling policies whereas dynamic controls result in adaptive scheduling policies. Real-time and hard-real time systems would benefit from dynamic race controls.

4. Centralized vs. Distributed Control

Control may be characterized by being centralized or distributed. The difference between these two forms of control is the point at which the decision is made, in the scheduler or in the program. Centralized control refers to the situation in which the program has **no** control over what should be scheduled next. There is no capability to directly resolve the races at the program level. In this case a scheduler exists that maintains exclusive control over all the races. The scheduler resolves the races using known scheduling policies which may be modified by monitoring events occurring in the system's environment. Distributed control refers to the situation in which the program has the capability to make decisions over what should be scheduled next. The programmer has the means to embed directives within the code to control the various races. Real-time systems could benefit from distributed control since the decision is made at the point of the race instead of passing information on to the centralized scheduler as to how to resolve the race.

5. An Analysis of the Control Types

The types of control discussed in the previous subsections can now be combined to characterize various schedulers. First, note that lack of control stands alone as a scheduling possibility. All control decisions are left completely to the compiler or runtime environment. This then leaves the eight scheduler possibilities listed in Figure 3. For each possibility, an example is given for its use in current languages.

As mentioned above static control leads to non-adaptive scheduling whereas dynamic control leads to adaptive scheduling. Combining this property with centralized or distributed control, schedulers are obtained with interesting characterizations. A static (non-adaptive) centralized scheduler has its scheduling policies *fixed at runtime and races would be resolved by the* scheduler, whereas a static (non-adaptive) distributed scheduler

SCHEDULER TYPES	EXAMPLES
Implicit - Static - Centralized	FIFO entry queues in Ada
Implicit - Static - Distributed	Ordered Alternative (PRI ALT) in Occam
Implicit - Dynamic - Centralized	No Language Equivalent
Implicit - Dynamic - Distributed	No Language Equivalent
Explicit - Static - Centralized	Task Priority in Ada
Explicit - Static - Distributed	Forerunner Control in Concurrent C (**by**)
Explicit - Dynamic - Centralized	*Task Priority in Concurrent C*
Explicit - Dynamic - Distributed	No Language Equivalent*

*Can be represented by the Explicit Comprehensive Set of Race Controls

Figure 3. Eight Scheduler Possibilities

allows the programmer to control the races but without any means of modifying that control at runtime. A dynamic (adaptive) centralized scheduler has the ability to change its scheduling policies during execution. The dynamic centralized scheduler would receive scheduling information from the executing program through the runtime system. The scheduler would then adapt its scheduling strategy to reflect the current state of the environment. On the other hand a dynamic (adaptive) distributed scheduler not only gives the programmer complete control of the races but also allows the programmer to adapt to the scheduling policy of the situation. Real-time reactive systems are enhanced with the ability to adapt to the environment. The Explicit Comprehensive Set of Race Controls is an example of a set of language constructs that supports dynamic distributed control.

V. THE CURRENT STATE OF RACE CONTROLS IN ADA

The current state of the race controls in Ada is as follows:

Priority Control
Ada has an explicit static priority control. Priority is implemented as a **pragma**; that is, as a suggestion to the compiler. Depending upon the compiler priority control may or may not be implemented. When a task type is defined and assigned a priority through the PRIORITY **pragma**, every task created of the same type will have the same priority. The scheduler schedules tasks in the order of their priority. There is no control for tasks that have equal priority.

Preference Control
Ada has no preference control. An arbitrary selection is made among open alternatives in the **selective wait**.

Forerunner Control
Ada has an implicit static forerunner control. Entry calls are serviced in a strictly FIFO manner.

Ada can be characterized as being weak and seriously lacking in the area of race controls. The concept of priority is not integrated consistently across all controls and there is no flexibility in the use of race controls. Control is performed by a static (non-adaptive) centralized scheduler. The result is that real-time control is virtually impossible.

The classification of controls by levels (program, task, and entry) in Section II gives a framework for understanding the dynamics of the Ada tasking model. Many of the problems and deficiencies that occur in the tasking model can be directly explained through the function of an individual race control or through the interaction between the various race controls. This classification scheme allows us to focus on each control as a separate entity wi [illegible] properties that affect the entire system in which it exists.

Due to the need for more sophisticated race controls, researchers and programmers have created Ada code workarounds to simulate the race controls necessary for their applications. Appendix B discusses methods for implementing dynamic priority control, static preference control and forerunner control. All of these simulations of the race controls have the common property that the programmer ends up fighting the language instead of taking advantage of it. The simulations of the race controls in Ada demonstrates a need for a more consistent, integrated approach to controlling races within Ada for real-time applications. Two proposals that respond to these needs are discussed in Sections VII and VIII.

VI. AN ANOMALY IN SCHEDULING ADA TASKS

One of the implications of scheduling real-time applications is that a high priority task is in more risk to meet its deadline than a low priority task [5]. One desires that the time that the high priority task is blocked from execution be minimized. Otherwise, urgent tasks may experience an unbounded delay at the expense of a less urgent tasks. A phenomenon has surfaced in the Ada tasking model whereby a low priority task is scheduled in lieu of a high priority task and has been termed **priority inversion** [4].

The intent here is not to claim that priority inversion has been rediscovered but to identify the deficiencies in the Ada tasking model design that caused the anomaly. Priority inversion occurs as a result of conflicts between the race controls. The interaction between race controls cause conflicts that cannot be resolved by the scheduler. This is due to the inconsistent definition of race controls in Ada: an explicit priority control, a lack of preference control and an implicit forerunner control. The Ada programmer should have the capacity to avoid priority inversion. Priority inversion is minimized by incorporating more deterministic race controls within the scheduler. Solutions to limit priority inversion must include scheduling controls that consider the interrelationships of the individual race controls and are consistent in their application. The Priority Inheritance Scheduler and the Explicit Comprehensive Set of Race Controls offer two viable solutions to minimize the effects of priority inversion. The main intent of the Priority Inheritance Scheduler is to minimize priority inversion; whereas the Explicit Comprehensive Set of Race Controls not only can minimize priority inversion but also can be used to implement a wide range of scheduling policies. Sections VII and VIII outline these control proposals and their application to the control of priority inversion. The following sections illustrate priority inversion resulting from priority control conflicts, priority-preference control conflicts, and priority-forerunner control conflicts.

1. Priority Control Conflicts

Program level scheduling produces a conflict in control in the following situation. Let three tasks T1, T2 and T3 have priorities P1, P2 and P3; respectively, where P1 > P2 > P3. If task T1 calls task T3 and T3 is not ready to accept the call, then T1 is delayed while T3 executes at a lower priority. This is acceptable since T1 requires T3's service. However, T1 can now be further delayed if task T2 gains the processor since its priority is higher than T3. Therefore, a lower priority task, T2, is executing in favor of the higher priority task, T1. This is a conflict in the intent of priority control. In Ada, there are no scheduling controls to prevent this situation. However, by identifying the problem within the context of priority control, it is clear that to resolve this anomaly, the system must be able to change the priority of its communication partner implicitly or explicitly. In this case, the priority of T3 needs to be raised higher than T2 to prevent T2 from gaining the processor.

2. The Priority Control - Preference Control Conflict

In Ada priority control is accomplished by the priority of the tasks; whereas there is no preference control. This conflict between program level scheduling and task level scheduling mechanisms causes priority inversion to occur at the task level. This results when an alternative with a lower priority entry call is selected over an alternative with a higher priority entry call. For example, suppose two tasks T1 and T2, where T1 has higher priority than T2, have called different entries of a task T3 within the same **selective wait** construct. If the alternative for which T2 is waiting gets selected first, then a lower priority task is scheduled in favor of a higher priority task. To eliminate this conflict, preference control must be consistent in nature with priority control; that is, preference control must exist at least in some form (static or dynamic) such that T1 is selected over T2.

3. The Priority Control - Forerunner Control Conflict

In Ada forerunner control is a static mechanism implemented by FIFO queues. The conflict between program level scheduling and entry level scheduling causes priority inversion at the entry level. This occurs when a lower priority entry call is accepted over a higher priority entry call to the same entry. Let task T1 has a higher priority than task T2 and both are calling the same entry of a task T3. If T2 has called the entry first, then it precedes T1 in the entry queue. T2 will be selected for rendezvous before T1; therefore, a lower priority task is scheduled in favor of a higher priority task. To eliminate this conflict, forerunner control must be implemented in such a way as to consider the priorities of the tasks waiting on the entry queue.

VII. THE SEI RACE CONTROL PROPOSAL FOR ADA

1. The Priority Inheritance Scheduler

Locke et al. [11] propose a Priority Inheritance Scheduler for Ada which controls all races through the use of a non-adaptive centralized scheduler. The Priority Inheritance Scheduler is characterized by the notion that Ada task priorities must be used whenever a choice is to be made within the tasking model. The components of a priority inheritance scheduler are:

1. **an explicit priority control,**
2. **an implicit preference control,** and
3. **an implicit forerunner control.**

Priority control is effected through the implementation of the priority inheritance protocol. The priority inheritance *protocol* [15] *specifies that the priority of a task will be* modified dynamically and implicitly to the greater of its own priority or the priority of the highest priority task waiting for it. The highest priority task is then selected for execution. Note that priority in the priority inheritance protocol is explicitly static but implicitly dynamic. The Ada 9X Working Group requirement (LI62) [12] for scheduling control is stronger since it requires a dynamic explicit priority control.

Preference control is accomplished implicitly by selecting the alternative with the highest priority entry call in its queue.

Forerunner control is accomplished implicitly by handling entry queues in the order of the caller's priority.

The Priority Inheritance Scheduler exhibited an average performance enhancement of about 33-50% over a standard Ada runtime scheduler in meeting deadlines. This integrated approach to race controls creates systems which are more predictable, portable and allow for reuse and easier maintenance.

The advantage of the Priority Inheritance Scheduler is that the user does not have to know about the detailed internal implementation. It is a simple tool to use. If the runtime systems supplies both the standard Ada scheduler and the Priority Inheritance Scheduler, the programmer has to decide which scheduler to enable. When the Priority Inheritance Scheduler is enabled, race controls are resolved as described above. When the standard Ada scheduler is enabled, race controls are resolved in accordance with the existing Ada rules.

The priority inheritance protocol is characterized as supportive of implicit control which is too rigid and not flexible enough for ongoing changing environments. There are limitations to the use of a Priority Inheritance Scheduler. It is tailored to a very restricted and specific need and thus is not general enough or versatile enough to deal with other scheduling algorithms. For example, the scheduler will only account for the relative priority of a task which does not always correlate to its criticality. When using the classic Rate Monotonic Scheduling Algorithm [10], the priorities are assigned with respect to the length of the task period and not with respect to task criticality to ensure that each task meets its deadline. The priority inheritance protocol can only be successfully implemented on a uniprocessor system. Hence programs written under the assumption of a uniprocessor are not portable to multiprocessors. In addition the priority inheritance protocol only accounts for modifying the priorities of called tasks. Therefore, priority inheritance could not be enforced when a high priority task is waiting for an entry call. In languages such as Ada that have an asymmetric naming convention, it is impossible to determine the identity of a potential caller.

2. Controlling Priority Inversion Via The Priority Inheritance Scheduler

The Priority Inheritance Scheduler implicitly controls all races through non-adaptive centralized scheduling. This integrated approach to race controls solves the problem of priority inversion.

Consider the control conflicts of Section VI using the Priority Inheritance Scheduler. For the priority control conflict (VI.1.), the priority inheritance protocol solves the problem. Task T3 would inherit the priority of task T1 and execute at that priority. T1 would only be delayed for the time of executing T3. Task T2 would not be scheduled since its priority would be less than T3's inherited priority. The priority inversion caused by the priority-preference control conflict (VI.2.) is eliminated due to the fact that the implicit preference control will select the alternative with the highest priority pending entry call. The priority-forerunner control conflict is resolved by the implicit forerunner control specifying that the entry call of the highest priority is selected.

VIII. A RACE CONTROL PROPOSAL FOR ADA

1. *The Explicit Comprehensive Set of Race Controls*

The Explicit Comprehensive Set of Race Controls [6] is the most versatile integrated set of race controls that can be applied to scheduling control. They represent a scheduling mechanism that is explicit, dynamic and distributed, well-suited for real-time reactive systems. This proposal is characterized by the notion that there should be a mechanism to explicitly control the choice whenever a choice is to be made within the Ada tasking model. The components of the Explicit Comprehensive Set of Race Controls are:

1. **an explicit priority control,** that includes the ability to assign and modify task priorities at runtime,
2. **an explicit preference control,** that allows the programmer to prioritize alternatives and alter the preference of alternatives at run time, and
3. **an explicit forerunner control,** that allows the programmer to prioritize entry calls not only by the priority of the caller but also by other parameters supplied by the caller.

These controls exist in the form of language constructs that can be applied anywhere within a program system. The Explicit Comprehensive Set of Race Controls provides an integrated, consistent approach to the resolution of all races.

Possible Ada language constructs expressing explicit preference control and explicit forerunner control are shown in Figure 4. ~riority, preference and forerunner control are incorpor as part of task specification to enforce the issue

```
type PRIORITIES is (TRIVIAL, IMPORTANT, CRUCIAL);
type PREFERENCES is (LOW, MEDIUM, HIGH);
type FORERUNNER is (SHORT_RANGE, MEDIUM_RANGE, LONG_RANGE);

task STRATEGIC_DEFENSE priority CRUCIAL is
     prefer HIGH: entry ATTACK(DISTANCE: in FORERUNNER)
                                   by (DISTANCE);
     prefer MEDIUM:  entry OFFENSE(...);
     prefer LOW:  entry DANGER(...) by (priority);
end STRATEGIC_DEFENSE;
```

Figure 4. Possible Priority, Preference and Forerunner Controls

that scheduling should be designed into an application. Task STRATEGIC_DEFENSE is assigned CRUCIAL priority for priority control. Other tasks within the program would be assigned a priority with respect to the desired scheduling policy at the program level. The key words "prefer" and "by" are used to implement preference control and forerunner control respectively. Within the task STRATEGIC_DEFENSE, an ATTACK message is accepted before any other messages regardless of the priority of callers to entry OFFENSE and entry DANGER. Calls to entry DANGER are accepted with the least preference. From all pending calls to entry ATTACK the call with the shortest distance is accepted first. DISTANCE is a parameter passed by the callers. Calls to the entry DANGER are accepted by the priority of the callers. Calls to the entry OFFENSE are accepted in standard FIFO order.

The Explicit Comprehensive Set of Race Controls represents a powerful scheduling mechanism that provides expressiveness, generality and portability (reusability) for real-time software development. Software system designers desire programs that accurately represent the problem domain. Programs can be developed more effectively when language structures are available that facilitate the transformation from the problem domain to the solution domain. This conforms to the software design principle [9] that minimizing the linguistic difference between the specification of the problem and the computer language not only minimizes the economics of the programming process, it also increases the reliability of the software that is written. The resultant programs are portable and independent of the runtime system since the controls are language features. In addition the Explicit Comprehensive Set of Race Controls shield the programmer from implementation details.

The Explicit Comprehensive Set of Race Controls can be used to realize a wide range of scheduling algorithms. Systems programmed in a language equipped with these controls are as predictable as the external environment. Control can be tailored to the specification of the application. The programmer may either choose to control a specific occurrence of a race or allow the scheduler to control the race.

Implicit centralized scheduling mechanisms, such as the Priority Inheritance Scheduler, can be built within the runtime system based upon scheduling rules dependent upon the type of system desired. The problem with these scheduling mechanisms is that programs can only be produced to conform with the targeted execution environment. The scope of application is limited since the application is dependent upon the execution environment.

2. Controlling Priority Inversion Via the Explicit Comprehensive Set of Race Controls

The Explicit Comprehensive Set of Race Controls have the power to avoid the problem of priority inversion and can be used to realize any scheduling algorithm. The effects of priority inversion can be limited satisfactorily by applying the Explicit Comprehensive Set of Race Controls according to the following procedures:

(1) A high priority task waiting for a rendezvous should explicitly request that a potential partner inherit its priority. This is a stronger than the priority inheritance scheduler as a high priority task waiting on an entry call may explicitly assign its priority to a potential caller. Due to the asymmetric naming convention in Ada, a potential caller is unknown to a priority inheritance scheduler. The capacity for explicit priority controls enables the inheritance of priority and eliminates control conflicts at the program level.

(2) The preference of an alternative must correlate to the priority of the pending call: the higher the priority of the pending call, the higher the preference of the alternative. As a result the alternative selected will be the one on which the waiting caller has the highest priority. The capacity for explicit preference control eliminates priority-preference conflicts at the task level.

(3) The forerunner control must select from the entry queue the task with the highest priority. The capacity for explicit forerunner control eliminates priority-forerunner conflicts at the entry level.

X. CONCLUSION

Ada represents a rich and powerful tasking model. However, nondeterminism exists within this model at the program level, the task level and the entry level creating an environment which is not conducive for real-time concurrent applications. Identifying nondeterminism by level suggests a classification scheme for scheduling controls of which Race Controls are the focus of this report. Race controls are important since their proper application enhances real-time programming. Race controls have been classified as Priority Control, Preference Control and Forerunner Control. Ada is weak and seriously lacking in the area of race controls. Simulations of the race controls in Ada demonstrate that the race controls are cumbersome for the programmer to apply within Ada applications. In addition, the phenomenon of *priority inversion* in Ada is *shown to exist* due to a conflict between the race controls.

Proposals to improve race controls in Ada include a priority inheritance scheduler and the Explicit Comprehensive Set of Race Controls. The Explicit Comprehensive Set of Race Controls include language constructs to explicitly control each race. This proposal is characterized by the notion that there should be a mechanism to explicitly control the choice whenever a choice is to be made. Possible language constructs for expressing preference control and forerunner control are demonstrated. The explicit set of race controls provides an integrated, consistent approach to resolution of all races. They are powerful enough to realize a wide range of scheduling algorithms and to avoid the problem of priority inversion.

XI. REFERENCES

[1] **Ada 9X Project Report**, Ada 9X Project Requirements Workshop, Office of the Under Secretary of Defense for Acquisition, Washington, D.C., June 1989.

[2] G. Booch, **Software Engineering with Ada**, The Benjamin/Cummings Publishing Company, 1983.

[3] A. Burns, "Using Large Families for Handling Priority Requests", **Ada Letters**, January/February, 1987, Vol. VII, No. 1.

[4] D. Cornhill, Task Session Summary, **Proceedings of the ACM International Workshop on Real-Time Ada Issues, Ada Letters**, 7(6), 29-32, 1987.

[5] D. Cornhill, L. Sha and J.P. Lehoczky, "Limitations of Ada for Real-Time Scheduling", **Proceedings of the ACM International Workshop on Real-Time Ada Issues**, May 1987.

[6] T. Elrad, "Comprehensive Race Controls for Ada Tasking", **2nd International Workshop on Real-Time Ada Issues, Ada Letters**, 8(7), 1988.

[7] N. Gehani and W.D. Roome, **The Concurrent C Programming Language**, Silicon Press, 1989.

[8] D. Harel et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", **Proceedings of the 10th International Conference on Software Engineering**, Singapore, April, 1989.

[9] M. Klerer, **User-Oriented Computer Languages**, MacMillan, New York, 1987, p. 11.

[10] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", **JACM**, Vol. 30, No. 1, January 1973.

[11] D. Locke, L. Sha, R. Rajkumar, J. Lehoczky and G. Burns, "Priority Inversion and Its Control: An Experimental Investigation", **2nd International Workshop on Real-Time Ada Issues, Ada Letters**, 8(7), 1988.

[12] Minutes of March 1, 1989, Ada Language Issues Working Group, **Ada Letters**, 9(4), 1989.

[13] A. H. Muntz and E. Horowitz, "A Framework for Specification and Design of Software for Advanced Sensor Systems", **Proceedings of the IEEE Tenth Annual Real-Time Systems Symposium**, December 5-7, 1989.

[14] L. Sha and J.B. Goodenough, "Real-Time Scheduling Theory and Ada", Technical Report, Carnegie-Mellon University, Software Engineering Institute, November 1988.

[15] L. Sha, R. Rajkumar and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", **Technical Report**, Department of Computer Science, CMU, 1987.

XII. APPENDIX A: LIST OF PUBLICATIONS, POSITION PAPERS AND PRESENTATIONS

T. Elrad, "Comprehensive Race Controls: A Versatile Scheduling Mechanism for Real-Time Applications", **Ada: The Design Choice, Proceedings of the Ada-Europe International Conference**, June 1989, Madrid, Spain.

T. Elrad and D.E. Nohl, "Reuse Concerns with Scheduling Dependent Concurrent Software", **Workshop on Language Issues for Reuse: Ada for the '90's**, September 26-29, 1989, Deer Isle, Maine.

T. Elrad and D.E. Nohl, "Concurrent Language Issues for Real-Time Scheduling Mechanisms", **Technical Report**, Illinois Institute of Technology, Chicago, Illinois, 1989.

T. Elrad and D.E. Nohl, "Race Controls for Portability and Adaptive Engineering", **Technical Report**, Illinois Institute of Technology, Chicago, Illinois, 1989.

T. Elrad and D.E. Nohl, "The Analysis and Comparison of Scheduling Controls in Concurrent Languages Through Classification", To be presented at the **21st SIGCSE Technical Symposium on Computer Science Education**, February 22-23, 1990, Washington, DC.

T. Elrad, "Comprehensive Race Controls for the Control of Nondeterminism in Ada", Presented at the **ALIWG**, August 1989, Ottawa, Canada.

T. Elrad, "A Position to the Ada 9X Requirements Workshop", May 22-26, 1989, Destin, Florida.

D. Levin, D. Nohl and T. Elrad, "A Clean Solution to the Readers-Writers Problem without the COUNT Attribute", To be presented at the **8th Annual National Conference on Ada Technology**, March 5-8, 1990, Atlanta, Georgia.

T. Elrad, Ada-9X Revision Request.

XIII. APPENDIX B: IMPLEMENTING THE RACE CONTROLS IN ADA

The following subsections give methods programmers and researchers have used to simulate race controls in Ada.

1. Priority Control

Ada lacks a **dynamic** priority control. Ted Baker in a memo to Tzilla Elrad outlined two methods to obtain dynamic priority control. One method involves the modification of the runtime system since priorities are already modified internally during rendezvous. One finds where the priority is stored in the task control block and allow it to be modified. This may involve the deletion of the task from the queue and reinserting it. The other method accomplishes dynamic priority control by creating a collection of **enabling** tasks of different priority levels. Tasks obtain their priority by rendezvousing with the appropriate enabling task. To change priority, the task must leave the **accept** statement and accept a call from another enabling task of a different priority. Neither solutions provide priority control that is useful and flexible enough for real-time applications. It is his opinion that anyone who desires dynamic priority in Ada either has the compiler vendor provide it or goes around Ada tasking completely.

2. Preference Control

Two methods are commonly used to simulate **static** preference control in Ada as illustrated in Figure 5. The first (Figure 5A) uses the attribute COUNT to prefer the entry calls in the order:

SERVICE1 (highest), SERVICE2, SERVICE3, and SERVICE4 (lowest). This preference simulation could fail when a calling task is aborted or the call timed out after the attribute COUNT is evaluated but before the rendezvous has started. The result would be an indefinite waiting period until another call of the same type occurs thus blocking all other calls. The second method (Figure 5B) makes use of nested select statements to enforce preference. The preference of entry calls is the same as in the previous case but in this case the preference simulation could result in costly and inefficient busy waiting if no current entry call exists. Preference also may not be given if a call to a higher preference alternative occurs after the alternative has been bypassed. Both solutions fail from a software engineering standpoint. They are unreadable and not easily modifiable if the preferences were to be changed.

3. Forerunner Control

Forerunner control can be simulated by using a family of entries [3] if the values within the range of priorities on which the entry calls are ordered are discrete and finite. For very small ranges, preference control using the attribute COUNT prioritizes the entry calls (Figure 6). This method suffers from the same drawbacks as discussed for the preference control solution using COUNT. As the size of the range increases, this technique quickly becomes unmanageable. For larger ranges, a loop can be used to run through all possible values within the family. This method could result in wasteful polling if the

```
(A)  select

          when B1
           ==> accept SERVICE1(..) do S1 end SERVICE1;
     or
          when B2 and (not(B1) or SERVICE1'COUNT = 0)
           ==> accept SERVICE2(..) do S2 end SERVICE2;
     or
          when B3 and (not(B1) or SERVICE1'COUNT = 0)
                  and (not(B2) or SERVICE2'COUNT = 0)
           ==> accept SERVICE3(..) do S3 end SERVICE3;
     or
          when B4 and (not(B1) or SERVICE1'COUNT = 0)
                  and (not(B2) or SERVICE2'COUNT = 0)
                  and (not(B3) or SERVICE3'COUNT = 0)
           ==>  accept SERVICE4(..) do S4 end SERVICE4;
     end select;

(B)  loop
       select
         when B1
           ==> accept SERVICE1(..) do S1 end SERVICE1;
               exit;
       else
         select
           when B2
             ==> accept SERVICE2(..) do S2 end SERVICE2;
                 exit;
         else
           select
             when B3 ==>
               accept SERVICE3(..) do S3 end SERVICE3;
               exit;
           else
             select
               when B4 ==> accept SERVICE4(..)
                                     do S4 end SERVICE4;
                           exit;
             else
               null;
             end select;
           end select;
         end select;
       end select;
     end loop;
```

Figure 5. Simulation of Preference Control in Ada.

entry queues are all empty. A reliable method that can be used for large ranges requires the use of two rendezvous (Figure 7). The first rendezvous (a call to ANNOUNCE) records that a task desires the service, passing its priority. The second rendezvous (a call to REQUIRE) initiates the service in priority order among the calling task requesting that service. This method also requires preference control so that a call to announce is preferred over a call for the service. Gehani and Roome offer a similar solution in [7].

There are assorted problems with this method. If there is an abortion of a client task between the call to ANNOUNCE and the acceptance of REQUIRE, the program could be delayed indefinitely. To avoid this, an anonymous agent task should be

```
type A_PRIORITY is (HIGH, MEDIUM, LOW);

task SERVER is
  entry REQUEST(A_PRIORITY)(...);
end SERVER;

task body SERVER is
begin
  loop
    select
      accept REQUEST(HIGH)(...) do ... end REQUEST;
    or
      when REQUEST(HIGH)'COUNT = 0  =>
        accept REQUEST(MEDIUM)(...) do ... end REQUEST;
    or
      when REQUEST(HIGH)'COUNT = 0 and
               REQUEST(MEDIUM)'COUNT = 0  =>
        accept REQUEST(LOW)(...) do ... end REQUEST;
    end select;
  end loop;
end SERVER;
```

Figure 6. Simulation of Forerunner Control in Ada using Preference Control (Very Small Ranges)

created by the interface procedure for each client task to make the calls to ANNOUNCE and REQUIRE on behalf of the client process. This method also requires a double rendezvous which leads to additional rendezvous overhead.

Sha and Goodenough [14] propose a method to handle entry calls in priority order by using a coding style that prevents queues from having more than one task. To accomplish this, though, a runtime system is needed that suspends tasks that call a server task already in a rendezvous. When the rendezvous is complete, the highest priority suspended task is activated and the entry call for that task is initiated.

```
generic
  type ELEMENT is (<>);
package TUPLE is
  function HIGHEST return ELEMENT;
  procedure ADD(E: ELEMENT);
  procedure REMOVE(E: ELEMENT);
  function NOT_EMPTY return BOOLEAN;
end TUPLE;

with TUPLE;
package RESOURCE is
  type SOME_PRIORITY is new INTEGER range 0..999;
  procedure REQUEST(P: SOME_PRIORITY; ... );
end RESOURCE;

package body RESOURCE is
  task SERVER is
    entry ANNOUNCE(P: SOME_PRIORITY);
    entry REQUIRE(SOME_PRIORITY)(...);
  end SERVER;

  procedure REQUEST(P: SOME_PRIORITY; ... );
  begin
    SERVER.ANNOUNCE(P);
    SERVER.REQUIRE(P)( ... );
  end REQUEST;

  package BIN is new TUPLE(ELEMENT => SOME_PRIORITY);

  task body SERVER is
    Po: SOME_PRIORITY;
  begin
    loop
      select
        ANNOUNCE(P: SOME_PRIORITY) do
          BIN.ADD(P);
        end ANNOUNCE;
      or
        terminate;
      end select;
      while BIN.NOT_EMPTY loop
        Po := BIN.HIGHEST;
        select
          accept ANNOUNCE(P: SOME_PRIORITY) do
            BIN.ADD(P);
          end ANNOUNCE;
        or
          when ANNOUNCE'COUNT = 0 =>
            accept REQUIRE(Po)( ... ) do
              ...
            end REQUIRE;
            BIN.REMOVE(Po);
        end select;
      end loop;
    end loop;
  end SERVER;
end RESOURCE;
```

Figure 7. Simulation of Forerunner Control in Ada. (Very Large Ranges)